

שפת Python



סיכום מלא ומתומנת
לשימוש כחומר עזר בבחינה

כתב וערך: אביחי לדון
2024



כמה מילימ' על הסיכום

סיכום זה נכתב בעיקר כדי לשמש מקור אינטואיטיבי, מותמצת ומסודר של שפת Python כחלק מחומר העזר לבחן בקורס "מבוא לתוכנות מערכות" בטכניון.

הסיכום מכיל את כל החומר שנלמד במסגרת הקורס ואף יותר, והוא מפורט בצורה עשיריה יותר עם תיאורים מילוליים (טקסטואליים) ודוגמאות של קטעי קוד מעוצבים.

מבנה הסיכום גובש לפי סדר הנושאים בהם רוכשים שפט תכונות חדשה, החל מיסודות השפה והתחביר הכללי שלה, ולכן קל מאוד להתמצא בו ולהසור זמן יקר לבחן.

תוכן הסיכום עצמו מתפרש על 33 עמודים בלבד. זאת לעומת 123 שkopיות מהמצגות של הקורס, שגם אם מסדר שתי שkopיות בכל עמוד, שזה המקסימום כדי שהתוכן ישאר קריין, נקבל מספר עמודים גדול כמעט פי 2 !

ممליץ להשייע ולהדפיס את הסיכום בדי צבעוני כדי להנוט מהקריאות של קטעי הקוד הצבעוניים, ולהמחשה:

```
print("This is how I look in B&W", end=' ' )  
print("And this is how I look in color", end=' ' )
```

שימוש מהנה ובהצלחה לכלם לבחן 😊

ניתן למצוא סיכומים נוספים ב-GitHub שלי בקישור: [avihayda.github.io/mySummaries](https://github.com/avihayda/mySummaries)

ליצירת קשר: dadonavihay@campus.technion.ac.il



תוכן עניינים

4	יסודות השפה – Python Basics	1
4	1.1 רקע כללי	
4	1.2 תחביר כללי	
5	1.3 הצעאת משתנים	
6	1.4 קבצי קוד Python – יבוא והרצה	
6	1.4.1 הרצה כתסריט	
6	1.4.2 Import – יבוא	
7	קלט ופלט – Input and Output	2
7	2.1 קלט	
7	2.2 פלט	
8	אופרטורים - Operators	3
8	3.1 אופרטורים אРИתמטיים	
9	3.2 אופרטורים השוואתיים	
9	3.3 אופרטורים לוגיים	
10	3.4 אופרטורי השמה	
11	3.5 אופרטורי זהות	
12	טיפוס נתונים – Data Types	4
13	4.1 מספריים – Numeric	
13	4.1.1 מספרים שלמים – Int	
13	4.1.2 מספרים ממשיים – Float	
13	4.1.3 מספרים מרוכבים – Complex	
14	4.1.4 המרחב בין הטיפוסי הנתונים המספריים	
14	4.2 בוליאניים – Boolean	
15	4.3 רצפים – Sequences	
18	4.3.1 רשימה – List	
19	4.3.2 מחרצת – String	
20	4.4 מילון – Dictionary	
21	4.5 קבוצה – Set	



23	בקורת זרימה – Control Flow	5
23	5.1 משפט התנאי <code>if</code>	
24	5.2 לולאות	
24	5.2.1 לולאת <code>while</code>	
24	5.2.2 לולאת <code>for</code>	
26	Comprehension 5.2.3	
28	פונקציות – Functions	6
28	6.1 העברת ארגומנטים	
29	6.2 פונקציות עם מספר פרמטרים משתנה	
30	מחלקות – Classes	7
30	7.1 בקרת גישה – Access Modifiers	
30	7.2 יושא במחלקות	
31	חריגות – Exceptions	8
31	8.1 האדרה ותחביר	
32	8.2 החריגות המובנות הנפוצות	
33	עבודה עם קבצים – File Handling	9
33	9.1 פתיחת וסגירת קובץ	
34	9.2 קריאה מתוך הקובץ	
34	9.3 כתיבה אל הקובץ	
34	9.4 קובץ JSON	
35	ספריות – Libraries	10
35	10.1 הספרייה os	
36	10.2 הספרייה sys	



1 יסודות השפה – Python Basics

1.1 רקע כללי

שפת Python היא שפת תכנות עילית המאפשרת שימוש בה לכמה עקרונות תכנות, כאשר אחד מהם זה שמוֹצָג במסגרת הקורס הוא תכנות מונחה- עצמים.

שפת Python היא **שפת תכנות מפורה**. בשונה משפות תכנות כמו שפת C/C++, שכדי להריץ את קבצי הקוד שלהם הם נדרשים לעבור הידור שמתרגם את הקוד לשפת מוכנה, שפת תכנות מפורה אינה עוברת הידור, אלא תהליך אחר בו קובץ הקוד עובר דרך תוכנה שנקראת "מפרש – Interpreter" אליו קובץ הקוד מוזן, שורה אחר שורה. המפרש, עברו כל פקודה בשפת המקור, מבצע אוסף מוגדר מראש של פקודות מוכנה. לכל אחד מהתהליכים ישנו יתרונות וחסרונות, אך לא נרחיב על כך. בנוסף, ניהול הזיכרון בשפת Python הינו אוטומטי.

בשפת Python קיימ סוג קובץ אחד^[1] והוא בעל הסיומת ".py".
בשונה משפת C/C++ בה קיימת חלוקה בין קבצי קוד – בעלי הסיומת .c ו-.cpp, לבין המיכלים ספריות – בעל הסיומת h (header), בשפת Python מטבחו שימוש עבור שני השימושים בקובץ ".py", ועל זאת נרחיב בהמשך הפרק.

1.2 תחביר כללי

בשונה משפות תכנות כמו C/C++, בה המהדר מתעלם מירידות שורה והזחות וישנו צורך בתווים מיוחדים כמו ";" ו-{ } כדי להגדיר את מבנה הקוד, בשפת Python מבנה הקוד מוגדר בעיקר על ידי השורות והזחות.

כל שורה מהוֹה פקודה אחת בלבד, ולכן אין צורך בתו מיוחד להפרדה.

במבנה של אחריהם מופיע בлок קוד (ולאלות, פונקציות וכו') נסמן בסוף שורת ההגדירה של המבנה את התו - ":" , שמצוין על פתיחת הבלוק.

נד שורה וביצוע לה הזחה, וכל השורות שמצוות באותו הרמה נחשבות כחלק מהבלוק.
הזהחה המקובלת בפתיחת בלוק בשפת Python היא בגודל של 4 רווחים, אך למעשה מספקת הזחה בגודל רווח אחד בשבייל זה. להלן דוגמה בסיסית להערכת התחביר:

```
x = 1 # no need to mention line separator character like ";"  
  
if x >= 1 : # no need to mention block boundary characters like "{}"  
    <statement> # Each indented line is considered part of the block  
  
x = 2 # first unindented line marks the end of the block
```

^[1] אמנים קיימות גם סיומות הבאות: ".pyd", ".pyo", ".pyw", ".pyc", ".pyc", ".py", ".pyw", ".pyd". אך הן אינן נלמדות במסגרת הקורס ולכן גם לא צוינו, וכיום אין סותר את הרעיון המועבר בפסקה זו.



בשפת Python, הקצאת המשתנים מתבצעת באופן פשוט למדי – בתחילת שורה חדשה נבחר שם למשנהו ונבצע אליו השמה עם הערך הרצוי. על שם המשתנה להתחיל מטא.

```
<new_variable> = <value>
```

ניתן להקצות ולחזור כמה משתנים באותה שורה, לדוגמה:

```
x, y = 1, 2
```

נשים לב כי אין צורך לציין את טיפוס המשתנה! וזאת מכיוון שבשפת Python למשנים אין טיפוס קבוע, אלא רק לערכים.

לכן ניתן לבצע השמה של ערך מכל טיפוס, גם אם כרגע הוא מכיל ערך מטיפוס מסוים אחר. לדוגמה:

```
x = 1 # x contains an integer  
x = "Hello" # x now contains a string
```

במידה ונרצה לדעת מה טיפוס הערך שהמשנה מכיל נשתמש בפונקציה `type` – המקבלת כารוגמנט את המשתנה ומהירה מחרוזת עם שם הטיפוס בפורמט הבא: `<class 'type_name'>`.
לדוגמה:

```
x = 1  
print(type(x)) # will print the string "<class 'int'>"
```

בדומה לשפות תכנות אחרות, ניתן גישה למשנה לא-קיים תגרום לשגיאת זמן ריצה ולהפסקת התוכנית.

ניתן להקצות משתנה "ריק" באמצעות אתחולו ל-`None`. מילת המפתח `None` משמשת כערך ריק.
לדוגמה:

```
x = None
```

- `None` אינו שקול ל-0, מחרוזת ריקה, `False` וכל מיני אובייקטים ריקים!
כיוון שכל אחד מהם הוא ערך תקין בפני עצמו, בזמן ש-`None` מייצג היעדר ערך.



1.4 קוצי קוד Python – "בוא והרצה

קיימות שתי דרכים לשימוש בקבץ קוד בשפת Python – הרצה – "תסריט – script" ו'יבוא. נציג בנפרד פירוט מלא על כל אחת מה דרכים.

הרצת כתסרייט 1.4.1.1

בשונה משפת ++C/C++ בה קיים מקום מוגדר ממנו התוכנית מתחליה להתבצע – הfonקצייה ()**main**, בשפת Python אין מקום מוגדר לכך. לכן, קובץ ה-**Chosper Python** אותו נבחר להריצ' – הוא יהיה זה שיקבע את "תסריט" התוכנית, כלומר ממנו התוכנית תתחילה ותסתתיים.

במידה ונרצה שהורות קוד מסויימות בקובץ שלמו ירצו אך ורק כאשר הקובץ מושך כתסריט, נכניס את אותן שורות הקוד לבlok הקוד הבא:

```
if __name__ == '__main__':
    statements
```

הרצת הקובץ כתסריט משורת הפקודה (h-shell) מתבצע באופן הבא:

PS C:\PythonSummary> python myFile.py

בהרצת הקובץ כתסריט משורת הפקודה ניתן גם להעביר ארגומנטים, אך פעולה זו דורשת שימוש בכלים יותר מתקדמים, ולכן נרchia על אפשרות זו בפרק 10.2.

Import - יבוא 1.4.2

כפי שכבר ציינו בתחילת הפרק, בשפת Python מtabצע שימוש בסוג קובץ יחיד. עם זאת, אין זה אומר כי علينا לרשום תוכנה שלמה בקובץ אחד, אלא שנוכל ליצור קבצים עם מימושים שונים ולהתיחס אליהם כל קובץ ספריה, בדומה לקבצי header משפט++/C/C++.

ואז בתחילתו של הקובץ הראשי של התוכנה – זה שיורץ כתסריט, נבצע יבוא שלם באמצעות המשפט `import` עם שם של הקבצים שיצרנו ובכך נוכל להשתמש במימושים שלהם בקובץ הראשי.

קובץ זה נקרא "モודול – Module".

"בוא קבצים מתבצע באופן הבא:

ישא myFile.py קובץ המכיל מימושים של פונקציות באופן הבא: `ch_function_1, function_2 ... function_n` כרך נבע יבוא מלא ושימוש במימושים (במקרה זה פונקציות) שבקובץ : myFile.py

```
import myFile # import the file/module  
myFile.function n() # n is the function number
```

ובძמלה ואנו לא זוגות כלם מושכים שבוגר אך רק למימוש פזיפי נוכל לבצע "בוא נסודת" בלבד:

```
from myFile import function_n  
function n() # direct access, without mentioning "myFile."
```



2 קלט ופלט – Input and Output

2.1 קלט

קלט מהמשתמש מתאפשר באמצעות הפונקציה `input` באופן הבא:

```
user_input = input(<prompt>) # prompt - optional, a string  
representing a default message before the input.
```

דוגמא לשימוש:

```
user_name = input("Enter your name: ")
```

הfonקציה מmirה את הקלט מהמשתמש ומחזירה אותו בתור מחוזת.

2.1.1 פלט

פלט מופק באמצעות הפונקציה `print` באופן הבא:

```
print(obj1, obj2, obj3, ...)
```

כאשר בירית המחדל היא שמודפס רווח אחד בין כל אובייקט, ובוסף של ההדפסה תבוצע ירידת שורה.
במידה ואנו מעוניינים לשנות הגדרות אלו נציג זאת כך:

```
print(obj1, obj2, obj3, sep = "our_separator" , end = "our_end")
```

במקרה של הדפסת ערך או משתנה של מספר עשרוני בו נרצה לקבוע את מספר הספרות אחרי הנקודה
שמודפסו, השתמש בפונקציה `print` באופן הבא:

```
print('%.#f' % <float_number>)
```

כאשר `#` הוא מספר הספרות המבוקש.



3 אופרטורים - Operators

3.1 אופרטורים אРИתמטיים

תחביר - Syntax	תיאור	אופרטור
$x + y$	חיבור	+
$x - y$	חיסור	-
$x * y$	כפל	*
x / y	חילוק	/
$x \% y$	מודולו – שארית החלוקת	%
$x // y$	חלוקת בשלים – ערך שלם תחתון של המנה	//
$x ** y$	חזקה – מחשב את האופרנד הראשון בחזקת האופרנד השני, כלומר את x^y	**



אופרטורים השוואתיים

3.2

תחביר – Syntax	תיאור	אופרטור
$x == y$	שווין בין ערכי המשתנים	$==$
$x != y$	שוני בין ערכי המשתנים	$!=$
$x > y$ $x < y$	גדול ממש קטן ממש	$<$, $>$
$x >= y$ $x <= y$	גדול או שווה קטן או שווה	$<=$, $>=$

אופרטורים אלו מוחזרים ערך בוליאני בהתאם לנוכנות הביטוי.

אופרטורים לוגיים

3.3

בשפת Python ישנו לשולה אופרטורים לוגיים, המשמשים לצירת ביטוי מורכב עבור משפט תנאי ולשלוב של כמה תנאים שונים תחת אותו ביטוי. להלן האופרטורים לפי סדר הקדימות:

1) **not** – כאשר האופרטור הלוגי **not** מופיע לפניו בביטוי במשפט תנאי, משפט התנאי יחזיר **True** כאשר הביטוי הכללי מחזיר **False**. משמש למקורה בו נרצה שתתבצע פעולה רק כאשר תנאי מסוים בהכרח **לא** מתקיים. זהה רענוןית לאופרטור הלוגי “!” בשפת C/C++.

2) **and** – אופרטור זה מופיע בין ביטויים ובכך מחברם לידי ביטוי אחד. הביטוי המאוחד יחזיר **True** אם ורק אם כל משפטי התנאי שמצדדיו מוחזרים גם **True**. במקרה ואחד מהביטויים יהיה **False**, הביטוי המאוחד יחזיר **False**. זהה רענוןית לאופרטור הלוגי “&&” בשפת C/C++.

3) **or** – בדומה לאופרטור **and** גם האופרטור **or** מאחד בין ביטויים, אך בשונה ממנו הביטוי המאוחד יחזיר **True** כאשר לפחות אחד מהביטויים שמצדדיו מוחזיר **True**. זהה רענוןית לאופרטור הלוגי “||” בשפת C/C++.

דוגמה לשימוש באופרטורים:

```
x, y = 2, 3
print(x>1 and y>1) # will print True
print(x>1 or y>1) # will print True
print(not x>1) # x>1 is True, so not True is False. will print False
```



3.4 אופרטורי השמה

בשפת Python ערכים מועברים "by reference", ולכן כאשר נבצע השמה של משתנה קיימן כלשהו למשתנה חדש, המשתנה החדש הוא למעשה רפנס של המשתנה המקורי.

בעקבות כך, אם ביצענו השמה שכזו, עלינו לקחת בחשבון שיתכן וערך המשתנה המקורי ישפיע משינוי שיבוצע במשתנה החדש - הרפנס.

בטיפוסים שאינם ניתנים לשינוי ("Immutable") כמו לדוגמה מספרים ומחרוזות, ביצוע שינוי יישנו בדרך כלל רק במשנה מקצת מקום חדש בזיכרון ובו הערך לאחר השינוי. וכך בטיפוסים שאינם ניתנים לשינוי אין חשש ממשינויים לא מבוקרים.

אך בטיפוסים הנитנים לשינוי ("Mutable"), כמו לדוגמה רשימות ו אובייקטים של מחלקות שיצרנו, החשש ממשינוי לא מבוקר קיים, שכן ביצוע שינוי ברפנס למשתנה מטיפוס זה יוצר שינוי גם במשתנה המקורי!
לשפת Python ישנו כמה כלים לייצרת עותקים "אמיתיים" של משתנים, שבהם אין חשש לשינויים לא מבוקרים, עליהם נרחב בפרק על טיפוסי נתונים.

תחביר – Syntax	תיאור	אופרטור
$x = 5$ $x = y$ $x = y + 5$	השמה של ערך, המשתנה או ביטוי אל תוך המשתנה.	=
$x @= y$ והוא שקול לביטוי $x = x$	נומן: @ - אופרטור אריתמטי (مالו שהגדכנו ב-3.1) הפעלת האופרטור האריתמטי @ בין האופרנדים והשמת התוצאה אל האופרנד השמאלי	@=

• האופרטורים האונאריים "++" ו"--" לא קיימים בשפת Python !



3.5 אופרטורי זהות

בשפת Python ישנו שני אופרטורי זהות: האופרטור `is` והאופרטור `in`.

- כאשר האופרטור `is` פועל בין אובייקט לבין ערך פועלתו תהיה לאופרטור `==`, כלומר תבצע השוואה בין הערך שבאובייקט לערך ובהתאם לתוצאה יוחזר ערך בוליאני.
כasher האופרטור `is` פועל בין שני אובייקטים, יבדק שוויון זהות ביןיהם, שבסונה מהאופרטור `==` בו השוויון בא לידי ביטוי בערך שבאובייקט בלבד, השוויון שיבדק הוא האם שני האובייקטים הם למעשה אותו אובייקט, כלומר, אם הם מייצגים את אותה כתובות בזיכרון. במקרים אחרות, האופרטור `is` בודק האם אחד האובייקטים הוא רפנסו לאובייקט השני. האופרטור `is` מחזיר ערך בוליאני בהתאם.

האופרטור `is` אינו סימטרי, כלומר ערך ההחזרה של הביטוי `"y is x"` זהה לשול הביטוי `"x is y"`.

- האופרטור `in` פועל בין אובייקטים וערכים, כאשר הוא מבצע בדיקה אם ערך או אובייקט כלשהו נמצאים בתוך אובייקט או ערך המציג מבנה נתונים כלשהו.
לכן, על האופרנד הימני ולהיות מתייפוס נתונים איטרטיבי. טיפוסי נתונים كالו נציג בפרק הבא.
האופרטור `in` מחזיר ערך בוליאני בהתאם.

דוגמה לשימוש באופרטורים:

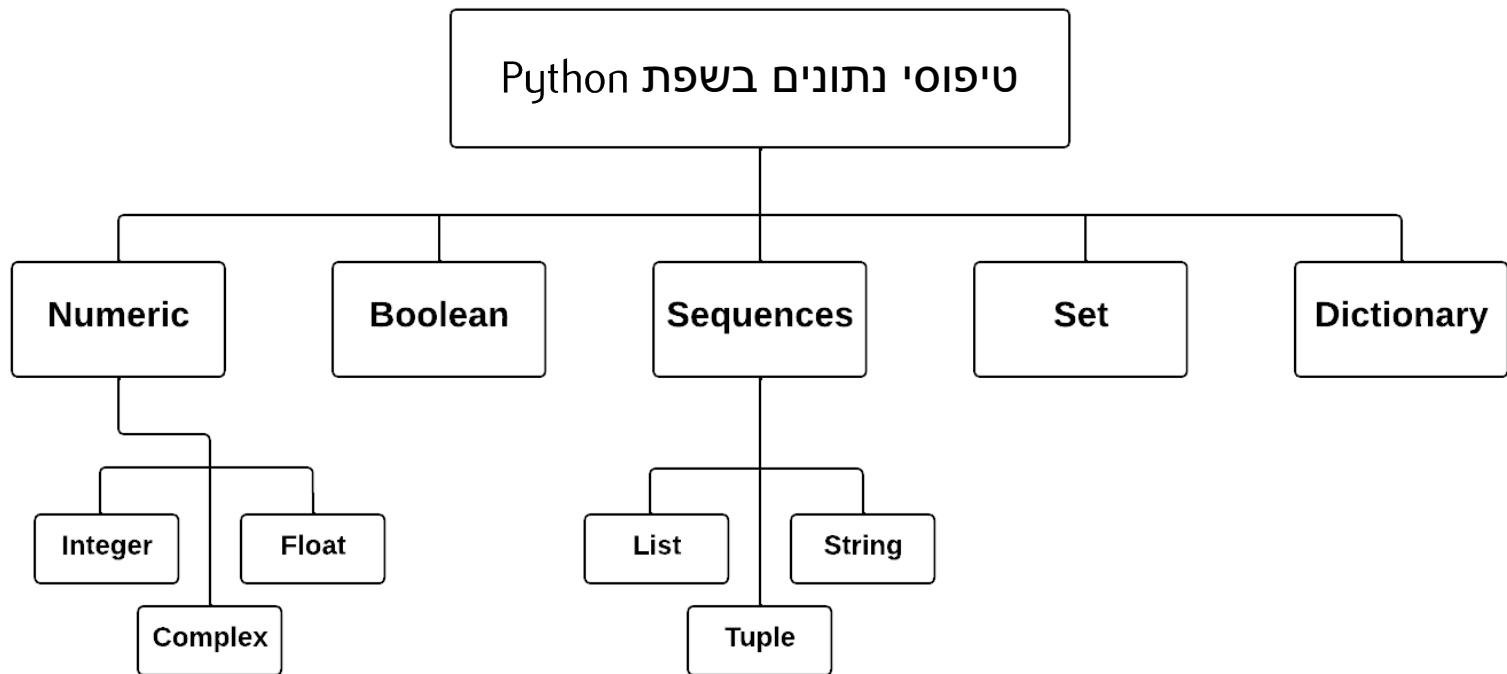
```
x = 1
print(1 is x) # will print True
y = x
print(x is y) # will print True
y += 1
print(x is y) # will print False, according to the explanation on 3.4

list = [1,2] # List type, iterable
print(1 in list) # will print True
print(x in list) # will print True, because 1 is in the list
print(1 in x) # will raise an exception, because x isn't iterable
```



4 טיפוס נתונים – Data Types

בפרק זה נציג את טיפוסי הנתונים המובנים בשפת Python.
להלן דיאגרמה ובה כל טיפוסי הנתונים:



במהלך הפרק נציג פירוט עבור כל אחד מטיפוסי הנתונים הנ"ל, ובו נביא את האופרטורים שניתן להפעיל עליו, את המethodות השימושיות שלו ופונקציות שניתן להפעיל על מופעיו.

. 5.2.3 Dictionary ו-List Comprehension • לא מוצג בפרק זה, ומובא במרקץ בפרק 3



4.1 מספריים – Numeric

בשפת Python ישנו 3 טיפוסי נתונים מספריים שונים:

- int – טיפוס לייצוג מספרים שלמים. קיצור של המילה Integer – מספר שלם.
- float – טיפוס לייצוג מספרים ממשיים.
- complex – טיפוס לייצוג מספרים מרוכבים.

4.1.1 מספריים שלמים – int

באמצעות משתנה מטיפוס int ניתן לייצג כל מספר ב- \mathbb{Z} – קבוצת המספרים השלמים. הקצהה של משתנה מטיפוס int תבוצע על ידי אתחול המשתנה למספר שלם כלשהו.

```
x = 1 # the type of x is int
```

4.1.2 מספריים ממשיים – float

בamuցuות משתנה מטיפוס float ניתן לייצג כל מספר ב- \mathbb{R} – קבוצת המספרים ממשיים. הקצהה של משתנה מטיפוס float תבוצע על ידי אתחול המשתנה למספר עם נקודה עשרונית כלשהו. אם נרצה משתנה מטיפוס float שערךו הוא מספר שלם a כלשהו, נאתחל את המשתנה למספר 0.a.

```
x = 1.2 # the type of x is float
```

```
x = 1.0 # x contains an integer, but the type of x is float
```

4.1.3 מספריים מרוכבים – complex

בamuցuות משתנה מטיפוס complex ניתן לייצג כל מספר ב- \mathbb{C} – קבוצת המספרים המרוכבים. הקצהה של משתנה מטיפוס complex תבוצע על ידי אתחול המשתנה באופן הבא:

```
x = a + bj # the type of x is complex
```

כאשר β , a הם ערכים מספריים מפורשים (אינם משתנים) מטיפוס int או float, ו- bj הוא הקבוע המרוכב.

הצבת משתנים (אם הם מכילים ערך מסוים) במקום a או β תוביל לשגיאת זמן ריצה שכתוכאה ממנה תזרק חריגה והתוכנית תעצר.
הקצאת משתנה מטיפוס complex על ידי משתנים תבוצע כמו בדוגמה הבאה:

```
x = 1  
y = 2  
z = complex(x,y) # z = 1 + 2j
```

ניתן לגשת לחלק ממשי ולחלק המדומה של המשתנה בנפרד, נדגים זאת על הדוגמה לעיל:

```
print(x.real) # will print 1.0  
print(x.imag) # will print 2.0
```



4.1.4 המרה בין הטיפוסים המספריים

המרה טיפוס משתנה מספרי מתבצעת על ידי הבנאים של כל אחד מהטיפוסים – `(int(), float())`.

ניתן לבצע המרה בין הטיפוסים `int` ו-`float`.

המרה מטיפוס `int` ל-`float` תקח את המספר השלם שבמשתנה מטיפוס `int` ותחזיר אותו עם 0. בסופו, וההמרה מטיפוס `float` ל-`int` תחזיר את החלק השלם שבמשתנה מטיפוס `float` על ידי השמתה הנקודה העשורתית והספרות שאחריה.

ניתן להמיר משתנה מטיפוס `int` או `float` לטיפוס `complex`, אך במקרה ההפוך לא ניתן. ההמרה למעשה מחליפה את הציגה המלאה או העשורתית של המספר בהציגה המרוכבת שלו, כלומר מספר זהה בחלק ממשי ו-0 בחלק המדומה (j^*).

```
x = 1
y = 2.1
z = int(y) # convert float to int, now z = 2
z = float(x) # convert int to float, now z = 1.0
z = complex(x) # convert int to complex, now z = 1+0j
z = complex(y) # convert float to complex, now z = 2.1+0j
z = int(z) # attempt to convert complex to int, will raise an error!
```

4.2 בוליאנים – Boolean

בדומה לשפות תכנות רבות, גם בשפת Python קיימים טיפוס הנתונים הבוליאני המיצג ערךאמת או שקר. אמת מיוצגת על ידי `True` ושקר על ידי `False`, עם דגש על האות הגדולה בתחילתן.

למעשה, האופרטורים הלוגיים שהגדכנו בפרק 3.3 הם אופרטורים בין משתנים או ערכים מטיפוס בוליאני.



רצפים היא כינוי לקבוצה של טיפוסי נתונים בשפת Python, שהמשותף להם הוא שם מתארים אוסף של אובייקטים מסוודרים בסדר מסוים. בשפת Python ישנו שלושה טיפוסי רצפים שונים:

- 1) List – רשימה מסודרת של איברים מטיפוסים שונים. דומה רעיון לסדרה מתמטית.
- 2) Tuple – ח-יה סדורה של איברים מטיפוסים שונים.
- 3) String – מחרוזת.

מציג אוטם בכלליות את התכונות המשותפות להן, ובהמשך נתמקד בכל אחת מהן ותכונות הייחודיות להן, למעט טיפוס הנתונים Tuple שכל תכונותיו مستcomes בתכונות המשותפות אותו מציג.

טיפוס הנתונים	תחביר – Syntax	ביצוע שינוי לאחר הקצהה
List – רשימה	<code>x = [obj1, obj2, ...]</code>	ניתן לשינוי Mutable
ח-יה סדורה – Tuple	<code>x = (obj1, obj2, ...)</code> עם איבר בודד: <code>x = (obj1,)</code>	בלתי ניתן לשינוי Immutable
מחרוזת – String	<code>x = "my string"</code> או <code>x = 'my string'</code>	בלתי ניתן לשינוי Immutable

מהות ההבדל בין List ל-Tuple הוא באפשרות לביצוע שינוי לאחר הקצהה.

אופרטורים

- "+ " – שרשור של רצפים מאותו סוג, כלומר `y + x = z` מוגדר אם `x` ו-`y` הם מאותו סוג של רצף.
- " * " – נקרא אופרטור הפיזור, ולמעשה מחזיר את איברי הרצף מופרדים באמצעות רווח יחיד. ניתן לומר שהוא "מפשט" את הרצפים. הפעלת האופרטור על רשימה או ח-יה סדורה תחזיר את איברי הסדרה כשרווח מפheid ביניהם, ללא הסוגרים והפסיקים. הפעלת האופרטור על מחרוזת תתייחס לכל תו כאיבר, ותחזיר את המחרוזת כאשר יש רווח יחיד בין כל תווים.
- "[start : end : step]" – אופרטור Slicing. משמש ליצירת עותק של הרצף כאשר `start` הוא האינדקס של האיבר הראשון (לא-כולל) ו-`end` הוא האינדקס של האיבר האחרון (לא-כולל) ו-`step` הוא הקפיצה באינדקסים מהם נבחר איברים לעותק. במידה ולא צינו `start` ו-`step` הפעולה תבצע על ערכי ברירת מחדל – `start = 0` ו-`step = 1`. במידה ונרצה את הרצף בסדר הפוך, נשתמש ב-Slicing באופן הבא: `[start : end : -1]`.
- שימוש ב-Slicing אינו משנה את אובייקט הרצף המקורי!



מתודות

הא **seq** רצף כלשהו.

- **seq.count(value)** – מוחזיר את מספר המופיעים של **value** ברכף.

במידה ו-**value** לא נמצא כל ברשימה תיזרק חריגה והתוכנית תיעצר!
לכן במידה ונתנו מעוניינים כי התוכנית תמשך משתמש במתודה תחת **try-except**.

פונקציות

הא **seq** רצף כלשהו.

- **len(seq)** – תחזיר את אורך הרצף.

• **min(seq, max)** – תחזיר את האיבר המינימלי/מקסימלי ברכף.
השימוש בפונקציות אלו מתאפשר רק אם כל איברי הרצף הינם מאותו הטיפוס, אחרת תיזרק חריגה!

- **tuple(seq), list(seq)** – בנאים של List ו-Tuple. נחלק למקירם:

(1) כאשר **seq** מטיפוס List – הפונקציה **tuple(seq)** תחזיר סדרה שאיבריה זהים לאיברי הרשימה **seq** ומוסדרים באותו סדר, והפונקציה **list(seq)** תחזיר את **seq** עצמו – שכן **seq** היא כבר רשימה.

(2) כאשר **seq** מטיפוס Tuple – הפונקציה **list(seq)** תחזיר רשימה שאיבריה זהים לאיבריה של הח-יה הסדרה **seq** ומוסדרים באותו סדר, והפונקציה **tuple(seq)** תחזיר את **seq** עצמו – שכן **seq** היא כבר ח-יה סדרה.

(3) כאשר **seq** מטיפוס String – הפונקציה **list(seq)** תחזיר רשימה קר שכל אות במחוזת **seq** היא איבר ברשימה כסדר האיברים הוא לפי מיקום האותיות במחוזת, והפונקציה **tuple(seq)** תחזיר ח-יה סדרה באותו האופן.

כמו שניתן לראות, שיטה זו אינה יעילה כאשר נרצה שאיברי הרשימה או הח-יה הסדרה יהיו מילוט המחוות.

במקרה זה נעדיף להשתמש במתודה **split** הייחודית למחוזות, שמחזירה רשימה שכל איבר בה הוא מילה במחוזות, המסודרת לפי סדר המילים במחוזות.
אין מתודה שמחזירה ח-יה סדרה באופן דומה אך אין צורך, כיוון שאנחנו יכולים באמצעות **split** לקבל רשימה אותה נהפוך בקלות לח-יה סדרה באמצעות **tuple()**.



Packing and Unpacking

כאשר אנו מזמנים ומאתחלים משתנה מטיפוס רצף, אנו למשה מבצעים פעולה שנקראת "Packing" – ככלומר "אזורדים" מספר איברים כלשהם אל תוך "מארך" כלשהו, כאשר ה"האזורדים" הם רשימות, ח-יות סדרות ומחוזות.

ק"י מת לכך גם פעולה הפוכה שנקראת "Unpacking", שמצויה את האיברים ש"ארצנו" מתוך ה"מארך". פעולה ה-"Unpacking" מתבצעת על ידי ביצוע השמה באזזה השורה של ה"מארך" – הרצף אל תוך מספר משתנים, שמספרם זהה למספר האיברים ברכף תיזרק חריגה והתוכנית עצה!

אם מספר המשתנים אינו זהה למספר האיברים ברכף תיזרק חריגה והתוכנית עצה!
ניתן לבצע "Unpacking" עם מספר משתנים קטן מאשר האיברים ברכף רק אם אחד מהמשתנים מגיע עם אופרטור הפיזור "**".

```
x = [1,2,3]
a,b,c = x
print(a,b,c) # will print 1 2 3
c, *d = x
print(c) # will print 1
print(d) # will print [2,3]
```



רשימה – List 4.3.1

כיוון שבונה מטיפוסי הרצפים האחרים, רשיימה היא טיפוס הנitin לשינוי (Mutable) קיימות לו מגוון מתודות ייחודיות שמרתתם לבצע מגוון שינויים ברשיימה קיימת.

מתודות ייחודיות

תהא **list** רשיימה כלשהי.

- `list.append(value)` – מוסיף את האיבר `value` לסוף הרשיימה.
- `list.insert(index, value)` – מוסיף את האיבר `value` למקום ה-`index` ברשיימה.
- `list.extend(otherSeq)` – משרשת לסוף הרשיימה רשיימה אחרת. עבור ארגומנט שהוא מחרוזת, תושירר לסוף הרשיימה `list` רשיימה זהה לרשיימה שתיצור (`otherSeq`).
- `(value)remove()` – מוחקת את המופיע הראשון של האיבר `value` ברשיימה.
ניסיין מחייבת איבר שאינו נמצא כלל ברשיימה תגרום לזריקת חריגה ועיצוב התוכנית!
לכן במידה אנחנו מעוניינים כי התוכנית תמשך נשתמש במתודה תחת `try-except`.
בנוסף, אם נרצה למחוק איבר לפי מקוםו ברשיימה נשתמש במלת המפתח `del` באופן הבא:
`del list[index]`
- `list.pop(index)` – מוציאה מהרשימה את האיבר במקום ה-`index` ומחזירה אותו.
- `list.copy()` – מחזירה עותק של הרשיימה עם מקום חדש בזיכרון.
כיוון שבשפת Python בירית המחדל היא שערכים מועברים על ידי reference, בעיות רבות עלולות להגרם בשימוש של טיפוסים הניטנים לשינוי כמו רשיימה, בעיות אותן הזכרנו בפרק 3.4.
אציג זאת בדוגמה הבאה:

```
x = [1, 2, 3]
y = x
y[0] = 6
print(x[0]) # will print 6 although we changed y, not x
```

למרות שביצענו שינוי ב-`y`, השינוי התרบצע גם על `x`.
ברוב המקרים לא נרצה תלות כזו אלא עותק נפרד בעל מקום אחר בזיכרון, ולשם כך נשתמש במתודה `copy`.
במידה וברשיימה לה נרצה ליצור עותק קיים איבר שהוא עצמו רשיימה, שינוי של תת הרשיימה בעותק ישפייע על תת הרשיימה המקורי, ולמצב זה המתודה `copy` אינה מספיקה.
נראה פונקציה המשמשת למצב שכזה.

פונקציות

- `list.deepcopy()` – מדובר בפונקציה `deepcopy` של המודול `copy`, שתפקידה הוא ליצור העותק של הרשיימה שהיא מקבלת, וגם ליצור העותקים גם לתתי-רישימות בתוך הרשיימה הראשית, ולמעשה היא הפתרון לבעה שהציגנו במתודה `(list.copy())`.
- המודול `copy` אמנם חלק מהספריות המובנות ולכן יש לייבא אותה באמצעות כתיבת השורה `import copy` בתחילת הקובץ.



Format Strings

באמצעות Format Strings נוכל ליצור מחרוזות שמשולבות בהן ערכים של משתנים בתוכנית. השימוש הכללי נראה כך:

```
str = f"content {obj1} content {obj2} content"
# for example
x = 5
print(f"The price of the product is: ${x}") # will print "The price
                                              of the product is: $5"
```

METHODS "יחודיות"

תזה str מחרוזת כלשהי.

- str.startswith(otherStr), str.endswith(otherStr) – בודקת אם המחרוזת str מתחילה או מסתיימת בתת-מחרוזת otherStr כלשהי ומחזירה ערך בוליאני בהתאם.
 - () – מוחקמת רווחים מתחילת ומסוף המחרוזת. דומה לפונקציה trim בשפת C++.
 - () – בודקת אם המחרוזת מורכבת ככל מהאותיות קטנות או גדולות באנגלית ומחזירה ערך בוליאני בהתאם.
 - () – בודקת אם המחרוזת מורכבת ככל מהאותיות באנגלית או מספרות ומחזירה ערך בוליאני בהתאם.
 - () – בודקת אם המחרוזת מורכבת ככל מתווי רווח (רווח, טאב, ירידת שורה) ומחזירה ערך בוליאני בהתאם.
 - otherString – מחזירה את האינדקס של המופיע הראשון של המחרוזת str בתוך str. במידה וotherString אינה תת מחרוזת של str המתודה תחזיר -1, וזאת בשונה מהמתודה () המשותפת לרצפים, שבמקרה זה הייתה זורקת חריגה וורמת לעצירת התוכנית.
 - (a, b) – יוצרת עותק של המחרוזת str בה כל מופיע של המחרוזת a הוחלף ב- b.
- ```
str = "AA"
print(str.replace("A", "B")) # will print "BB"
```
- () – מחזירה רשימה (List) שכל איבר בה הוא מילה במחרוזת, המסדרת לפי סדר המילים במחרוזת. הגרסה המלאה של המתודה היא str.split(separator, maxsplit), כאשר separator הוא התו שמספריד בין המילים והוא רווחים כברירת מחדל, ו-maxsplit הוא מספר השילובים.



בשפת Python מילון הוא מבנה נתונים המגדיר אוסף של מפתחות - Keys וערכים - Values, המורכב ממיפוי חד-ערך בין מפתח לערך.

המילון דומה רעיון ל-Map בשפת C++, אך בשונה ממנו במילון אין הגבלה על הטיפוס של המפתחות והערכים. יצירת מילון ויתחולו מתבצעת באופן הבא:

```
myDictionary = {key1 : value1, key2 : value2, ...}
myDictionary = {} # empty dictionary
```

הוספה זוג חדש של מפתח-ערך או עדכון ערך של מפתח קיים לאחר האתחול הראשוני מתבצע כך:

```
myDictionary[new_key] = new_value # add a new key-value pair to the dictionary
myDictionary[key1] = updated_value # This will update the value of key1
```

ניתן לגשת לערך באמצעות המפתח שלו באופן הבא:

```
x = myDictionary[key1] # x contains the value of key1
```

כדי לבדוק אם אובייקט כלשהו הוא מפתח במילון השתמש באופרטור הזיהות `in` שהגדכנו בפרק 3.5. לדוגמה:

```
print(key1 in myDictionary) # This will print True
```

## מתודות

יהא `dict` מילון כלשהו.

- `()() dict.keys()`, `dict.values()` –מחזירה אובייקט "view" איטרטיבי, שמהווה מעין "חלון" ומאפשר להציג את איברי המפתחות/ערכים של המילון.

`()() dict.items()`מחזירה אובייקט "view" שמאפשר להציג את איברי המילון כזוג סדר (`key, value`).

- `(key) dict.pop()` – מקבלת מפתח כารוגמנט, מחזירה את הערך המתאים לוותה המפתח ומוחקת מהמילון את אותו הזוג.

במידה ולא קיים במילון מפתח מתאים לזה שנshall כארוגמנט תזרק חריגה והתוכונית תיעצר!  
לכן במידה ואני מעוניינם כי התוכנית תמשך נשתמש במתודה תחת `try-except`.  
דרך נוספת למחיקת איבר באמצעות מילת המפתח `del`.

- `() dict.clear()` – מוחקת את כל הזוגות במילון כך שלאחר מכן המילון ריק.

## פונקציות

יהא `dict` מילון כלשהו.

- `len(dict)` – מחזירה את מספר הזוגות שבמילון.



## 4.5 קבוצה – Set

בשפת Python קבוצה היא מבנה נתונים המכיל אוסף לא-סדר של ערכים, כאשר כל ערך יכול להופיע בקבוצה פעמי אחת בלבד. זהה רעוניות לקבוצה מתמטית. קבוצה יכולה להכיל ערכים מטיפוסים שונים, ובתנאי שהם מטיפוס המוגדר כבלתי ניתן לשינוי (Immutable).

כמפורטה מכך שקבוצה היא אוסף לא-סדר היא אוסף ללא אינדקסים, אך למרות זאת קבוצה היא מבנה נתונים איטרטיבי, וישנם דרכם לעבור על איברי הקבוצה אותן נציג בפרק זה. יצירת קבוצה חדשה ויתחוללה מתבצעת באופן הבא:

```
set = {obj1, obj2, ...}
to create an empty set:
set = set() # that because "set = {}" will create an empty dictionary
```

- בשפת Python קבוצה לא יכולה להכיל אובייקט שהוא עצמו קבוצה ! כיון שקבוצה לא יכולה להכיל ערכים מטיפוסים הנתינים לשינוי, כאשר טיפוס הנתונים Set ניתן לשינוי (Mutable) .

### אופרטורים

- " | " – אופרטור האיחוד, מוחזיר את הקבוצה המתקבלת מהאיחוד - ס של שתי הקבוצות.
- "&" – אופרטור החיתוך, מוחזיר את הקבוצה המתקבלת מהחיתוך - ח של שתי הקבוצות.
- " - " – אופרטור ההפרש, מוחזיר את הקבוצה המתקבלת מההפרש - \ של שתי הקבוצות.
- " ^ " – אופרטור ההפרש הסימטרי, מוחזיר את הקבוצה המתקבלת מההפרש ההסימטרי – ־� של שתי הקבוצות.
- ">=", "<=" – אופרטור לבדיקת הכליה. בודק אם קבוצה מוכלת/מוכלת-שווה לקבוצה אחרת ומחזיר ערך בוליאני בהתאם.
- ניתן להשתמש בכל אחד מהאופרטורים לעיל, למעט האחרון, בשילוב עם השמה – "=" , "|=" , "&=" , "-=" , " ^= " . קיימות גם מתודות לביצוע פעולות אלו, אך השימוש באופרטורים נוח יותר.

### מתודות

תפקיד `set` קבוצה כלה!.

- (`set.add(value)` – מוסיף את הערך `value` לקבוצה. אם הערך `value` כבר שייך לקבוצה לא יתבצע דבר.
- (`set.remove(value)` – מסירה את הערך `value` מהתוכנית ! ניסיין הסרת ערך שאינו נמצא בקבוצה תגרום לזריקת חריגה ועצירת התוכנית !
- (`set.discard(value)` – מסירה את הערך `value` מהתוכנית, אך בשונה מרחתודה `remove`, אם הערך אותו ניסינו להסיר לא נמצא בקבוצה לא יתבצע דבר והתוכנית תמשך.
- (`set.copy()` – מוחזירה עותק של הקבוצה בעל כתובות חדשה בזיכרון.
- (`set.clear()` – מוחקת את כל איברי הקבוצה כך שלאחר מכן הקבוצה ריקה.



## פונקציות

תזהה `set` קבוצה כלה.

• – תחזיר את מספר האיברים בקבוצה `len(set)`.

• – תחזיר את האיבר המינימלי/מקסימלי ברכף `min(set), max(set)`.

• – תחזיר רשימה ממויינת של איברי הקבוצה בסדר עולה `sorted(set)`.

**השימוש בפונקציות אלו מתאפשר רק אם כל איברי הרצף הינם מאותו הטיפוס, אחרת תיזרק חריגה!**

• (`sum(set)` – אם `set` היא קבוצת מספרים, הפונקציה תחזיר את סכום איברי הקבוצה.

**במידה ואחד מאיברי הקבוצה אינם מספר, תיזרק חריגה והתוכנית תעצר!**



## 5 בקרת זרימה – Control Flow

בשפת Python, בדומה לשפות תכנות רבות, בקרת הזרימה מתבצע בעיקר באמצעות שני מבני בקרה:

- לולאות – לולאת `while`, לולאת `for`
- משפט/פקודת תנאי – משפט התנאי `if`

החל מגרסת 3.10 של שפת Python שוחררה בשנת 2021, התווסף לשפה משפט תנאי חדש בשם-`match-case` זהה למשפט התנאי "switch-case" המוכר לנו משפת C/C++. אך מכיוון שבמסגרת הקורס אנו עובדים עם גרסה 3.6, משפט תנאי זה אינו נלמד ולכן לא נרחיב עליו.

### 5.1 משפט התנאי `if`

הרכבת משפט התנאי `if` תתבצע באופן הבא:

```
if <condition>:
 statements
elif <condition>: # optional, same as "else if" from C/C++
 statements
else: # optional
 statements
```

בנוסף, קיימ גם אופרטור טרינארי המאפשר להגיד שתי פעולות ומשפט תנאי, כך שאחת הפעולות תتمמש בהתאם לתוצאה של משפט התנאי, וכל זאת בשורה אחת בלבד:

```
<statement_1> if <condition> else <statement_2>
for example: (x is a numeric variable)
print("x is greater than 3") if x > 3 else print("x is less than 3")
```

- האופרטור מוגדר לצמד `if-else` בלבד, וכן לא ניתן להשתמש בו עם הצמד `if-elif`.



## 5.2 לולאות

בשפת Python קיימים שני סוגי של לולאות – לולאת `while` ולולאת `for`.  
בדומה לשפות תכנות אחרות, מוגדרים המשפטים `continue` ו- `break` המאפשרים שליטה בזרימת הלולאה.

- גורם לסיום האיטרציה הנוכחיית ויציאה מהלולאה.

- – גורם לסיום האיטרציה הנוכחיית וממשיך ישר לאיטרציה הבאה של הלולאה.

ברוב המקרים נמקם משפטיים אלו בתוך משפט תנאי כאשר ביטוי התנאי יהיה צהה שאם הוא מתקיים נרצה לסיים את האיטרציה והኖchet, ואז להמשיך לאיטרציה הבאה או לצאת מהלולאה ולהמשיך בתוכנית.

### 5.2.1 לולאת `while`

הגדרת לולאת `while` תתבצע באופן הבא:

```
while <condition>:
 statements
```

- **הלולאה do לא קיימת בשפת Python !**

### 5.2.2 לולאת `for`

בשפת Python ישנה גישה שונה בימוש לולאת `for`, צו שכביר בשלב הגדרת הלולאה מצהירים על איזה אוסף נרצה לעבור, ובאופן אוטומטי מגדרה את תנאי העצירה שלה לגודל האוסף.  
גישה זו נוחה וקלת, משפרת את הקריאה ובטוחה מבחינה זכרון.  
הגדרת לולאת `for` תתבצע באופן הבא:

```
for i in <iterable>:
 statements
```

כasher:

- **<iterable>** הוא ערך או משתנה מטיפוס נתונים המהווה אוסף איטרטיבי.  
כל האוספים המובנים בשפת Python – List, Tuple, String, Dictionary – ו- Set עוניים על הגדרה זו.

- **i** הוא שם המשתנה המקובל בכל איטרציה של הלולאה ערך של פריט אחר מהאוסף.  
ניתן להחליף את האות **i** בכל אות או מילה אחרת שנחפוץ.

הלולאה תעריך כאשר היא סימנה לעבור על כל פריטי האוסף, כך שלמעשה תנאי העצירה שלה הוא כאשר מספר האיטרציות גדול ממספר הפריטים באוסף.  
ישנו עוד כמה גרסאות לולאת `for` בשפת Python העושות שימוש בפונקציות המובנות בשפה.  
נציג פונקציות אלו ואת הגרסאות לולאת `for` שהן יוצרות.



## הפונקציה range

היא פונקציה מובנית בשפת Python, המקבלת מספר שלם `ch`, ומחזירה אוסף איטרטיבי עם `ch` פריטים החל מ-0 עד `ch` בסדר עולה. נשתמש בפונקציה `range` באופן הבא:

```
myRange = range(stop) # the basic form, will return sequence from 0 to 9
myRange = range(start, stop, step) # start and step are optional
```

כאשר:

- `start` הוא הערך ההתחלתי של הרץ' שהפונקציה תחזיר. ערך ברירת המחדל שלו הוא 0.
- `stop` הוא הערך בו מסתיים הרץ' (לא כולל).
- `step` הוא ההפרש בין כל איבר ברצף. ערך ברירת המחדל שלו הוא 1.

נניח ואני רוצים להציג לולאת `for` "כלאotic" – זו הדומה ללולאת `for` בשפת C/C++ בה אנו מגדירים שירוט את מספר האיטרציות שנרצה שהיא תבצע. כך נעשה זאת בקלות באמצעות הפונקציה `range`:

```
for i in range(stop):
 statements
```

בלולאה זו יתבצעו "stop" איטרציות, ו- `i` מקבל במהלך האיטרציות כל ערך שלם בתחום (1 – [0, `stop` – 1].

## הפונקציה enumerate

היא פונקציה מובנית בשפת Python, המתקבלת `<iterable>` כלשהו ומחזירה אוסף איטרטיבי של זוגות-סדרים (`Tuple` של שני איברים) בפורמט הבא: (`<iteration_number>, <item>`), שבעל איטרציה `i` מקבל את הזוג המתאים למספרה. ניתן גם להויס משטנה נוסף לצד `i` ואז כל אחד מהם יקבל שירוט את ערכי הזוג-הסדר.

במידה ונרצה לדעת בכל איטרציה מה מספרה, שעבור טיפוסינו נתונים מסווג רצף מספר זה הוא גם האינדקס של אותו פריט, נשתמש בפונקציה `enumerate` באופן הבא:

```
for i in enumerate(iterable): # i contains a tuple
 statements
for i,e in enumerate(iterable): # i contain the index and e contains the item
 statements
```



## Comprehension 5.2.3

היא דרך מהירה וקצרה ליצור אוסף חדש מຕוך אוסף איטרטיבי קיים, וזאת על ידי מעבר על כל פריטי האוסף באמצעות לולאת `for`, עליה נפרט בפרק הבא, והגדרת פעלת חישוב שתופעל על כל פריט לפני שיכנסו לאוסף החדש.

קיים גם אפשרות החישוב, בנוסף לפעולות החישוב, להוסיף לרשימה החדש רק פריטים מסוימים מתוך האוסף וזאת ע"י ביטוי תנאי, כך שפריטים שבביטוי התנאי יჩזרו ערך `False` לא יכנסו לרשימה.

ניתן להשתמש ב- Comprehension כדי ליצור אוספים מטיפוס הנתונים הבאים:

- Dictionary – Dictionary
- Set – Set
- List – List

שימוש ב- Comprehension ליצור רשימה וקבוצה מתבצע באופן הבא:

```
new_list = [<expression> for x in <iterable>] # List Comprehension
new_set = {<expression> for x in <iterable>} # Set Comprehension
optional - with condition
new_list = [<expression> for x in <iterable> if <condition>]
new_set = {<expression> for x in <iterable> if <condition>}
```

כasher:

- **<expression>** הוא ערך או משתנה מטיפוס נתונים המהווה אוסף איטרטיבי שמננו ליצור את האוסף החדש. כל האוספים המובנים בשפת Python – List, Tuple, String, Dictionary – ו- Set עונים על הגדרה זו.
- **<iterable>** הוא ביטוי חישובי כלשהו על **x**, שמקבל את פריט מהאוסף בכל איטרציה.  
[!] יש לוודא כי כל פריטי הרשימה מוגדרים לפעולות חישוב זו, אחרת תיזרק חריגה והתוכנית תיעצר!
- **<condition>** הוא ביטוי תנאי כלשהו שנרצה לבדוק על **x**.

אם נמ אין הגדרה רשנית ל- Comprehension, אך אם אין צורך מכיוון שניתן למש את הרעיון שלו בקלות על ידי יצירת רשימה עם List Comprehension והמרתה לח-יה סדרה עם הבנייה (`tuple`).

שימוש ב- Comprehension ליצור מילון מעט שונה ותבצע באופן הבא:

```
new_dict = {<key_exp> : <value_exp> for x in <iterable>}
optional - with 2 iterables:
new_dict = {<key_exp> : <value_exp> for x,y in zip(<iter1>, <iter2>)}
optional - with condition:
new_dict = {<key_exp> : <value_exp> for x in <iter> if <condition>}
```

כasher:

- **<key\_exp>** ו- **<value\_exp>** הם ביטויים חישוביים על **x** או על **x** ו- **y** ליצור key-value למילון.
- **<iter1>** ו- **<iter2>** הם שני **<iterable>** שונים. נשתמש בתוכונה זו כאשר נרצה ליצור את המפתחות מתוך אוסף אחד ואת הערכים שלהם מתוך אוסף אחר.
- בדף הבא, נציג כמה דוגמאות לשימוש ב- Comprehension.



## דוגמה 1

ניקח רשימה מספרים כלשהי וניצור רשימה חדשה עם ריבוע האיברים במקומות הזוגיים:

```
list = [1,2,3,4]
new_list = [x**2 for x in list if x % 2 == 0]
print(new_list) # will print the list "[4, 16]"
```

## דוגמה 2 + הפקציה `isinstance`

נמחיש באמצעות דוגמה זו את האזהרה שציינו לעיל<sup>1</sup> בהקשר של הפעלת הביטוי החישובי על פריטי האוסף.

```
set = {1, 2, "Hi"}
new_list = [x+1 for x in set]
```

בדוגמה זו `set` היא קבוצה המכילה מספרים ומחרוזת. כשהלאלה מגיע אל המחרוזת `"Hi"` היא תנסה לבצע את פעולה החישוב `1 + "Hi"` שאינה מוגדרת, ולכן תזרק חריגה והטכנית עצה. נוכל להמנע מבעיות כאלה על ידי שימוש בביטוי תנאי שיוודא תחילת אמ טיפוס הנתונים של הפריט הוא צה שמוגדר לפעולות החישוב, ובהמשך לדוגמה שנותנו:

```
set = {1, 2, "Hi"}
new_list = [x+1 for x in set if isinstance(x, int)]
```

כאשר (`obj, data_type`) היא פונקציה מובנית בשפת Python שמקבלת ארגומנט אובייקט ושם טיפוס נתונים, בודקת אם האובייקט הוא מהטיפוס שהתקבל ומהירה ערך בוליאני בהתאם. בבירור טיפוס נתונים של אובייקט נעדייף לבצע בה שימוש הפקציה (`type(obj)`), זאת מכיוון שהפקציה `isinstance` תומכת בירושה ותחזיר `True` במקרה והאובייקט הוא ממולקה יורשת של טיפוס הנתונים שנשלח. בנוסף, בארגומנט של טיפוס הנתונים היא מקבלת גם ח-יה סדרה של שמות טיפוס נתונים והיא תחזיר `True` אם האובייקט הוא מופיע של אחד מהם.

## דוגמה 3

```
first_names = ['Avihay', 'Daniel', 'Aviv']
last_names = ['Dadon', 'Klein', 'Censor']
names_dict = { f : l for f,l in zip(first_names, last_names)}
```

בדוגמה זו מתבצע Dictionary Comprehension לייצרת מילון בו התארים והשמות הפרטיים הם המפתחות ושמות המשפחה הם הערךם, כאשר אנו משתמשים בפעולה חישובית על מנת לשרשר את התואר לשם הפרט. אחד הזוגות במילון שנוצר הוא: `{'Dr Aviv' : 'Censor'}`.



## 6 פונקציות – Function

בשפת Python, בניגוד לשפות אחרות, אין צורך לציין את טיפוס ערך ההחזרה של הפונקציה או אם היא בהכרח מוחזירה ערך, וכןנו כן קר גם לגבי טיפוס הפרמטרים שהפונקציה מקבלת. בנוסף, אין צורך לבצע הכרזה אלא רק הגדרה ומימוש, וניתן להשתמש בה החל מהשורה שמתוחת למימושה. הדבר היחיד שצריך לציין לפני שם הפונקציה אותה נרצה ליצור הוא את המילה `def` – קיצור של `define`. הכרזה על פונקציה מתבצעת באופן הבא:

```
def function_name (parm1, parm2, ...):
 statements
 return expression # optional
```

ובאופן מורחב ומלא:

```
def function_name (parm1 : <data_type>, ...) -> <return_type>:
 statements
 return expression # optional
```

כאשר `<data_type>` וה `<return_type>` הם המלצה לטיפוס הארגומנט ולטיפוס ערך ההחזרה של הפונקציה. הגדרה זו אינה מחייבת והפונקציה לא תזרוק חריגה במידה ויתקבל או יוחזר ערך מטיפוס שונה מהה שצוין. כמו כן, המלצה על טיפוס ערך ההחזרה אינה מהוות חובה על החזרת ערך, אך שניתן למשמש את הפונקציה קר שהיא לא תחזיר ערך.

### הערות חשובות:

- ערך ההחזרה של פונקציה שלא מכילה פקודת `return` הוא הקבוע `None`.
- הפקודה `pass` מצינית פקודת ריקה, ובמיצועה נוכל להגדיר פונקציה ריקה מבלי שתתקבל שגיאה.
- בשפת Python אין העמסת פונקציות.
- ניתן להעביר פונקציה כפרמטר לפונקציה אחרת באמצעות שם הפונקציה.

### העברה ארגומנטים 6.1

העברה ארגומנטים לפונקציה בשפת Python מתחבצע בשתי דרכים עיקריות:

1. לפי סדר (Positional Arguments) – העברת הארגומנטים בהתאם לסדר הופעתם בחთימת הפונקציה.

2. לפי שם (Named Arguments) – העברת הארגומנטים על ידי שימוש לשם הפרמטרים שהוגדרו בפונקציה. בדרך זו לכל ארגומנט משוייר לפרמטר, ולכן אין חשיבות לסדר הארגומנטים.

בנוסף, ניתן להגדיר בפונקציה פרמטרים עם ערכי ברירת מחדל (Default Arguments) וזאת במקרה שבבקיריה לפונקציה לא העברו הערכים שלהם כารוגומנטים, אותן פרמטרים יקבלו את ערכי ברירת המחדל.

להלן דוגמה אחת הממחישה את כל השלושה – Positional, Named and Default Arguments

```
def helloMessage(name, title = 'Mr./Ms.'):
 print(f"Hello {title} {name}! How are you?"")

helloMessage('Aviv', 'Dr.') # Positional arguments
helloMessage(title = 'Dr.', name = 'Aviv') # Named arguments
both will print: "Hello Dr. Aviv! How are you?"
helloMessage('David') # Default parameter value
```



## 6.2

### פונקציות עם מספר פרמטרים משתנה

ניתן להגדיר פונקציה שמקבלת מספר לא-ידיוע מראש של פרמטרים, וזאת באמצעות הוספת “\*” לשם הפרמטר בחתימת הפונקציה, וכל הארגומנטים שיועברו לפונקציה יועברו כח-יה סדורה (Tuple) של ערכים. ניחוש באמצעות דוגמה כיצד זה מתבצע:

```
def my_function (*params):
 for i in params:
 print(i)

my_function(1, 2, 3, 4, 5)
```

כל הספרות שנשלחו כארוגומט לפונקציה יועברו כח-יה סדורה ובה הספרות לפי סדר המיקומים, ככלומר מתקיים כי `(5, 1, 2, 3, 4) = params`.

קיים דרך נוספת, בה נסיף ”\*\*” לשם הפרמטר, נעביר את הארגומנטים לפונקציה בזוגות, ובכך הם יועברו כמילון (Dictionary) של ערכים. ובממש לדוגמה:

```
def my_function (**params):
 for i in params.items():
 print(i)

my_function(key1 = 1 , key2 = 2, key3 = 3)
```

כל הזוגות שנשלחו כארוגומנט לפונקציה יועברו כמילון ובירו הזוגות שנשלחו הם מפתחות וערכים. ככלומר מתקיים כי `{'key1': 1, 'key2': 2, 'key3': 3} = params`.



## 7 מחלקות – Classes

בדומה לרוב המוחלט של שפות תכנות מונחות-עצמיים, גם שפת Python עשויה שימוש עיקרי במחלקות. להלן כמה נקודות על מחלקות בשפת Python:

- אין צורך להזכיר בmphorsh על שדות המחלקה.
- הגישה לשדות ולMETHOD של המחלקה היא `public` כבירית בלבד.
- זיכרון, ניהול היזכרון בשפת Python הוא אוטומטי ולין אין צורך בהורס – `destructor`.
- בניית המחלקה הוא METHOD בשם `__init__` ( `"_" x 2` לפני ואחרי `in` )
- כל בניין או METHOD של המחלקה יוגדר עם `"self"` כפרמטר, כך שלכל בניין או METHOD יש לכל הפחות פרמטר אחד. `"self"` מבצע תפקיד דומה ל- `"this"` משפט C++, ורק באמצעותו יוכל לגשת לשדות המחלקה.

**יצירת מחלקה מתבצעת באופן הבא:**

```
class MyClass:
 def __init__(self, arg1, ...): # constructor
 self.myClassField = name
 .
 .
 def MyClassMethod(self):
 <statement>
```

### 7.1 בקרת גישה – Access Modifiers

ניתן לנצל את הגישה לשדות וMETHOD של מחלקה באמצעות שימוש בתו קוו תחתון - `"_"` (Underscore). כדי להזכיר כי שדה או METHOD היא privateünk לעילנו לציין קוו תחתון כפול בתחילת שמה, למשל `"__name"`. עבור שדה או METHOD נציין קוו תחתון יחיד, למשל `"_name"`.

- אין אפשרות להציג שדה `private` או `protected` להגדרת המחלקה, ולכן ניתן להציגם אך ורק בהגדירה הראשונית של המחלקה. כל שדה שנוצר מוחוץ למחלקה, גם אם נוסיף לתחילה שמו `"_"` או `"__"`, הוא שדה `public`.

### 7.2 ירשה במחלקות

במידה ונרצה ליצור מחלקה יורשת למחלקה אחרת נציין זאת בשורה הראשונה:

```
class derivedClass(baseClass):
```

כך ש- `derivedClass` היא מחלקה יורשת של מחלקה `baseClass`.



## 8 חריגות – Exceptions

8.1 הגדרה ותחביר

בשפת Python, שימוש בחיריגות מתרחש באמצעות שלושת מילוט המפתח הבאות:

1. `raise` – מילת המפתח לזריקת החיריגה, מקביל למילת המפתח "throw" בשפת C++.

2. `try` – בתוך הבלוק של `try` יהו שורות הקוד מהם יתכן ונקבל חיריגה.

3. `except` – תופיע לאחר הבלוק ה- `try`, ולאחריה בלוק ובו שורות קוד שנרכזה שייתבצעו אם אכן תזריך חיריגה מהבלוק שמתוחת לו `try`.

להלןקטע קלשוי בקוד בו קיימת זריקת חיריגה:

```
if <condition>:
 raise <exception>()
 : try :
 statements # some statements that may raise an exception
try:
```

ומיד לאחר הבלוק של `try` נרשם את הקוד שיפעל במקרה שתזריך חיריגה במסגרת `try`:

```
except <exception>:
 statements # some statements that will handle the exception
```

כאשר `<exception>` היא חיריגת ספציפית אותה נרצה "لتפое".  
ניתן לתפое כמה חיריגות בו זמנית באמצעות הבא:

```
except (<exception1>, <exception2>, ...): # tuple of exceptions
```

במידה ולא נציג חיריגת ספציפית – כל חיריגת תפое.

לאחר הבלוק `except` ניתן להוסיף את הבלוק `else` שירוץ אם לא נזרקה חיריגה, ואת הבלוק `finally` שירוץ בין אם נזרקה חיריגה ובין אם לא, כאשר שימוש נפוץ בו הוא כדי לסגור קבצים.

```
: Exception
class myExceptions(Exception):
 pass
```

זריקת חיריגה באמצעות `raise` יכולה להתבצע עם העברת ארגומנטים בתוך הסוגרים שללאחר שם החיריגה.  
נציג אפשרות זו באמצעות דוגמה:

```
if x != y:
 raise ValueError(x)
```

כדי לתפое את החיריגה עם הארגומנט עליינו למש את `except` באופן הבא:

```
except ValueError as e:
 print(f"Error: {e} is not a valid value")
```

כאשר המשתנה `e` מקבל את ערכו של `x`.



## 8.2 החריגות המובנות הנפוצות

בשפת Python ישן חיריגות שМОובנות בה, ולמעשה למעלה מ-30. להלן רשימה של עשרת החריגות המובנות הנפוצות:

| סיבת הזריקה                                                  | שם החריגה                |
|--------------------------------------------------------------|--------------------------|
| העברת ערך לא-חוקי לפונקציה או מתודה                          | <b>ValueError</b>        |
| ניסיון גישה לפרט באוסף עם אינדקס שאינו קיים                  | <b>IndexError</b>        |
| ניסיון גישה למפתח שאינו קיים במלון                           | <b>KeyError</b>          |
| ביצוע פעולה לא-חוקית בין שני טיפוסים נטוניים שונים           | <b>TypeError</b>         |
| שימוש בשם משתנה שאינו מוגדר                                  | <b>NameError</b>         |
| ניסיון לחלק מספר באפס                                        | <b>ZeroDivisionError</b> |
| ניסיון גישה לשדה פרטוי או מתודה שאינה קיימת באובייקט         | <b>AttributeError</b>    |
| פעולה אשר אין מספיק זיכרון זמין כדי לבצע אותה                | <b>MemoryError</b>       |
| בעיה בפעולות קלט/פלט בעבודה עם קבצים                         | <b>IOError</b>           |
| טיפוס נתוניים מסוימים שתוצאה פעולה חישובונית גדולה מדי עבורו | <b>OverflowError</b>     |



## 9 File Handling – עבודה עם קבצים

### 9.1 פתיחת וסגירת קובץ

לפנינו שוכן לבצע שינויים בקובץ, נדרש תחילה לבצע לקובץ פתיחה. פתיחת קובץ תבוצע על ידי הפקציה `open` באופן הבא:

```
my_file = open(<file_name>, <mode>)
```

כאשר `<mode>` הוא אחד משלושת המצביעים הבאים:

- `'r'` – מצב קרייה בו תתאפשר קריאה מהקובץ בלבד. מצב קרייה הוא ברירת המחדל ולא צוין מצב.
- `'w'` – מצב כתיבה בו תתאפשר כתיבה לקובץ בלבד. במידה והיה בקובץ תוכן קודם, הוא ימחק מיד לאחר פתיחת הקובץ, גם אם בפועל לא נבצע שום כתיבה.
- `'a'` – מצב כתיבה אך בשונה מ-`'w'`, במידה והיה בקובץ תוכן קודם הוא לא ימחק, והכתיבה לקובץ תבוצע אל סוף התוכן הנוכחי.

וכאשר `<file_name>` הוא אחד משלושת המצביעים הבאים:

- נתיב (path) כלשהו לקובץ במערכת הקבצים.
- שם מלא כולל סימנת של קובץ הנמצא בתיקיית הפרויקט (התיקייה בה נמצא קובץ הקוד).
- במצב כתיבה – `'w'` בלבד, `file_name` יכול לשמש שם של קובץ חדש אותו אנו רוצים ליצור. במידה ורשمنו שם קובץ שלא נמצא בתיקיית הפרויקט, התוכנית תיצור קובץ חדש בשם זה ותפתח אותו במצב המבוקש.

בסוף הקריאה או הכתיבה יש לבצע לקובץ סגירה. השארת הקובץ פתוח במהלך התוכנית תגרום לעומס מיותר בזיכרון ותמנע גישה אליו מתוכניות אחרות, ואם ביצענו כתיבה לקובץ זה יוביל להתנהגות לא-צפויה וייתכן שהתוכן שכתבנו לא ישמר. סגירת קובץ תבוצע על ידי המתודה `close` באופן הבא:

```
my_file.close()
```

הדרך המומלצת והבטוחה לפתיחת וסגירה של קבצים בשפת Python היא באמצעות שימוש במנגן `with`, שבאמצעותו נפתח את הקובץ, וונרשם את הפעולות שנרצה לבצע בבלוק שמתחתיו. בסיום הבלוק הקובץ יסגר באופן אוטומטי. שימוש במנגן `with` מתבצע באופן הבא:

```
with open(<file_name>, <mode>) as my_file:
 <statements>
```



## 9.2 קריאה מתוך הקובץ

לאחר שפתחנו את הקובץ, נוכל לבצע בו שינויים בהתאם למצב בו הוא פתוח.  
קריאה מתוך הקובץ תבוצע באמצעות 2 מетодות עיקריות:

- `my_file.read()` – מחזירה מחרוזת אליה היא קראה את כל הקובץ.
- `my_file.readline()` – מחזירה מחרוזת אליה היא קראה שורה אחת מהקובץ.
- `my_file.readlines()` – מחזירה רשימה של מחרוזות, כאשר כל מחרוזת ברשימה מייצגת שורה אחת שנקראה מהקובץ.

## 9.3 כתיבה אל הקובץ

כתיבה אל הקובץ תבוצע באמצעות המתודה:

- `my_file.write(str)` – מקבלת מחרוזת וכותבת אותה בקובץ.
- המתודה אינה מבצעת ירידת שורה בסוף הכתיבה, לכן במידת הצורך יש להוסיף למחוזת تو ירידת שורה.  
הוספה تو ירידת שורה למחוזת יכולה להתבצע על ידי שרשור تو ירידת שורה - ' \n ' לסוף המחרוזת.

## 9.4 קובץ JSON

JSON (ראשי תיבות של JavaScript Object Notation) הוא פורמט קובץ קל להעברת מידע  
שימוש בטקסט קרי-אלדום כדי לאחסן ולהעביר מבני מידע המורכבים מזוגות של מפתח-ערך.  
אף על פי שפורטט זה פותח בתחילת השימוש בשפת התכנות JavaScript, JSON הוא מבנה נתונים לא תליי  
שפה, ותמייה בו מבנית במגון שפות תכנות וביניהן שפת Python.

הספרייה הסטנדרטית של שפת Python מכילה את המודול `json` שבו פונקציות לקריאה וכתיבה של מיליוןים  
לפומט JSON בצורה פשוטה ומהירה. פונקציות אלו הן:

- `(indent = 4, <json_file>, <my_dict>) = json.dump(<my_dict>)` – מקבלת מילון וקובץ JSON  
פתחו במצב כתיבה ומבצעת כתיבה של המילון לקובץ, כאשר `indent` הוא פרמטר גודל ההזזה.  
במידה ולא נציג את גודל ההזזה, ערך ברירת המחדל שיתקבל הוא 0 – כולם ללא הזזה.  
לכן, כדי לקבל קובץ JSON מעוצב וקרייא עליון לציג את גודל פרמטר ההזזה.
- `(<json_file>.load(<json_file>)) = json.load(<json_file>)` – מקבלת קובץ JSON פתוח במצב קריאה ומחזירה מילון עם המידע  
מקובץ ה-JSON.
- כדי להשתמש בפונקציות אלו עלינו ליבא את המודול בכר שנצין בתחילת הקוד "import json".



## 10 ספריות – Libraries

ספריות בשפת Python הן אוסף של מודולים – קבצי קוד וביהםימושים שונים אותם נוכל ליבא לכל תכנית. ישן ספריות שモוגנות בשפת Python, כמו הספרייה הסטנדרטית שהזכרנו בסוף הפרק הקודם. ספריות שאינן מוגנות יותקנו בשורת הפקודה באמצעות מערכת ניהול הספריות דוקן. קיימים מגוון רחב של ספריות המוגוונות את היכולות של השפה. הן מכוסות מגוון רחב של נושאים, כולל עיבוד טקסט, פעולות מתמטיות, גישה לקבצים ולספריות, שימוש מידע, ועוד. ספריות הן תוצר של עבודה הרבה ובדיקות מקיפות והשימוש בהן מבטיח קוד איקוטי ויעיל, וכן יכולת נועדי לשימוש בספריה כדי מת מאשר לכתוב את הקוד בעצמו.

### 10.1 הספרייה os

הספרייה os בשפת Python היא ספרייה המכילה מגוון רחב ועוצמתי של כלים המאפשרים אינטראקציה עם מערכת הפעלה כמו ניהול קבצים ותיקיות, מידע ופיקודות מערכת מתוך הקוד. נציג כמה פונקציות בסיסיות מהספרייה os לעובדה עם קבצים:

• היא `<path>` נתיב כלשהו במערכת קבצים.

- (`os.listdir(<path>)`) – מקבלת נתיב לתיקייה כלשהו ומוחזירה רשימה של מחרוזות עם שמות הקבצים שבתיקייה. במידה ולא ישך נתיב ארגומנט לפונקציה הנתיב יקבל ערך ברירת מחדל של נתיב התיקייה שבה נמצא קובץ הקוד.  
**הערה: נתיב ישיר לקובץ ארגומנט לפונקציה תגרום לזריקת חריגה ועיצוב התכנית!**

- `sep`.os – תחזר את התו המשמש כפריד בין רכיבים בנatie קבצים במערכת הפעלה הנוכחיות.

- (`join(*<path>, <path>)`) – מקבלת מחרוזות המייצגת רכיבים לנatie קבצים ומוחזירה נתיב מלא המסדר לפי סדר הרכיבים, עםתו פריד השיר למערכת הפעלה הנוכחיות.

- (`dirname(<path>)`) – מקבלת נתיב כלשהו ומוחזירה נתיב لتיקיית האב.

- (`path.basename(<path>)`) – מקבלת נתיב כלשהו ומוחזירה את שם הקובץ או התיקייה שנמצאים ביעד של הנתיב שהתקבל.

- (`split(<path>)`) – מקבלת נתיב כלשהו ומוחזירה זוג-סדור זהה לזוג-הסדר שיתקבל על ידי: `(path.dirname("(<path>")`, `os.basename("(<path>")`).

- (`path.splitext(<path>)`) – מקבלת נתיב לקובץ כלשהו ומוחזירה זוג-סדור של הנתיב לקובץ ללא הסיומת שלו, והסיומת שלו. במידה והקובץ בלי סיומת, או שהנתיב שהוא עבר ארגומנט לפונקציה הוא נתיב لتיקייה, הפונקציה תחזיר זוג-סדור כשהاء הנתיב והאייר השני הוא מחרוזת ריקה.

- (`isfile(<path>)`) – מקבלת נתיב כלשהו, בודקת אם יעד הנתיב הוא קובץ ומוחזירה ערך בוליאני בהתאם. קיימת גם הפונקציה `isdir` שהיא בעצם ה- `not` של `isfile`.



## 10.2 הספרייה sys

הספרייה `sys` בשפת Python היא ספרייה המספקת מגוון רחב של כלים המאפשרים אינטראקציה עם המפרש (interpreter) של Python ועם מערכת הפעלה. היא כוללת פונקציות וובייקטיבים לניהול פרמטרים של שורת הפקודה, טיפול בזרמי קלט ופלט סטנדרטיים, ניהול נתיב החיפוש של מודולים, וטיפול בשגיאות וחריגות מערכתיות.

נזכיר שימוש בסיסי וחשוב בספרייה `sys` שנלמד חלק מהקורס – קליטה וניהול הארגומנטים המתקבלים משורת הפקודה לתוכנית שלנו בעת הרצת הקובץ כתסריט, באמצעות הפונקציה `sys.argv[num]`.

להלן פקודת ההריצה כתסריט משורת הפקודה:

```
PS C:\PythonSummary> python myScript.py file1 file2
```

נשתמש בפונקציה `sys.argv[num]` כדי לקלוט את הארגומנטים בקובץ הקוד שלנו באופן הבא:

```
import sys

obj1 = sys.argv[1] # obj1 = file1
obj2 = sys.argv[2] # obj2 = file2
```

- בדומה למה שראינו בשפת C/C++, `sys.argv[0]` מכיל מחורזת עם שם הקובץ, וכן נקלט ארגומנטים החל מהאינדקס 1.